Using VLSI to Reduce Serialization and Memory Traffic in Shared Memory Parallel Computers

Susan Dickey, Allan Gottlieb, Richard Kenner[1]

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

Ultracomputer Note #94

January, 1986

ABSTRACT

The NYU Ultracomputer is an architecture for a large scale MIMD (Multiple Instruction stream, Multiple Data stream) shared memory parallel computer that may be viewed as a column of processors and a column of memory modules connected by a rectangular network of enhanced two by two buffered crossbars. The primary novelty of the design is the ability of the network to combine multiple requests directed at the same memory location, including a new coordination request, fetch-and-add. This permits task coordination to be achieved in a highly parallel manner. For example, if an arbitrary number of tasks simultaneously issue inserts or deletes to a single shared queue that is neither empty nor full, then all these inserts and deletes are accomplished in essentially the same time as would be required for a single insert or delete.

This report reviews the Ultracomputer architecture and system design and describes the VLSI enhanced buffered crossbars that are the key to the highly parallel behavior mentioned above.

Consider a powerful machine composed of thousands of processors and gigabytes of memory. With 10-20 MIPS (including floating point) and a megabyte of memory soon to be available on a dozen chips, such a configuration could be built to yield significantly more performance than current supercomputers with roughly the same component count. Moreover, due to replication of parts, the

[1]This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under contract number DE-AC0276ER03077, and in part by the National Science Foundation, under grant number DCR-8413359. This paper will appear in *Advanced Research In VLSI: Proceedings of the Fourth MIT Conference*, Charles E. Leiserson, editor.

number of distinct components would be quite small.

Hardware design and assembly of a multiprocessor with a very high degree of parallelism therefore poses no new problems. However, actually using all the processing power that can theoretically be generated presents a two-fold challenge. First, several thousand processors must be coordinated in such a way that their aggregate power is applied to useful computation. Serial procedures in which one processor works while the others wait become bottlenecks that drastically reduce the power obtained. The cost of serial bottlenecks rise linearly with the number of processors involved; in any highly parallel architecture, they must be eliminated. Second, the machine must be programmable by humans. High degrees of parallelism require simple languages and easy-to-use facilities for designing, writing, and debugging parallel programs in order to be effectively used.

Our group has proposed [5] that the hardware and software design of a highly parallel computer should meet the following goals.

- Scaling. Effective performance should scale upward to a very high level. Given a problem of sufficient size, an n-fold increase in the number of processors should yield a speedup factor of almost n.
- General purpose. The machine should be capable of efficient execution of a wide class of algorithms, displaying relative neutrality with respect to algorithmic structure or data flow pattern.
- Programmability. High-level programmers should not have to consider the machine's low-level structural details in order to write efficient programs. Programming and debugging should not be substantially more difficult than on a serial machine.
- Multiprogramming. The software should be able to allocate processors and other machine resources to different phases of one job and/or to different user jobs in an efficient and highly dynamic way.

Achievement of these goals requires an integrated hardware/software approach. The burden on the system designer is to support a high-level and flexible programming model to enable the development of software which schedules the processors so that the workload is balanced, and, most importantly, avoids critical sections that would constitute unacceptable serial bottlenecks.

The next section reviews the MIMD shared memory computational model on which the Ultracomputer is based. As indicated below, this model is not realizable in hardware. In particular the postulated single cycle access to shared memory must be weakened to permit access via a processor to memory interconnection network having logarithmic latency. A hardware design closely approximating this model is sketched in section three with the interconnection network described in the subsequent section. Section five discusses selected issues in the design of the custom VLSI switches to be used in this network. In particular, the return path (i.e. the memory to processor direction) design is presented. We conclude with a brief report of the current status and future plans of our VLSI effort.

1. Ultracomputer Architecture

In this section we review the architectural model on which the Ultracomputer is based, and discuss its power. Although this idealized machine is not physically realizable, a close approximation can be built. Elements of the actual machine design are described in order to illustrate integrated hardware/software mechanisms for bottleneck-free coordination of a very large number of processors.

1.1. The Model

An idealized parallel processor, dubbed a "paracomputer" by Schwartz [18] and classified as a WRAM by Borodin and Hopcroft [1], consists of a number of autonomous processing elements (PEs) sharing a central memory (see also [6,19]). Every PE is permitted to read or write a shared memory cell in one cycle. In particular, simultaneous reads and writes directed at the same memory cell are effected in a single cycle.

In order to make precise the effect of simultaneous access to shared memory we define the *serialization principle*, which states that the effect of simultaneous actions by the PEs is as if the actions had occurred in some (unspecified) serial order. Thus, for example, a load simultaneous with two stores directed at the same memory cell will return either the original value or one of the two stored values, possibly different from the value which the cell finally comes to contain. Note that, in this model, simultaneous memory updates are in fact accomplished in one cycle; the serialization principle speaks only of the effect of simultaneous actions and not of their implementation.

In an actual hardware implementation, single cycle access to globally shared memory is not possible to achieve. For any technology there is a limit, say b, on the number of signals that one can fan in at once. Thus, if N processors are to access even a single bit of shared memory, the shortest access time possible is $\log_b N$. As will be seen, hardware achieving this logarithmic access time has been designed, but cannot use off the shelf components. A custom VLSI design is needed for switching components in the processor to memory interconnection network. In addition to increasing the design time, the network adds to replication cost and size. For a fixed number of dollars (or cubic feet, or BTUs, etc.), such a shared memory design, achieving the minimum memory access time, will contain fewer processors or memory cells than will a strictly private memory design constructed from the same technology and requiring no additional components for the communication network.

Although we believe that the lower peak performance inherent in shared memory designs is adequately compensated for by their increased flexibility and generality, this issue has not been settled. Most likely the answer will prove to be so application dependent that both shared and private memory designs will prove successful.

1.2. The Fetch-and-add Operation

We augment the paracomputer model with the "fetch-and-add" (F&A) operation, a powerful interprocessor synchronization operation that permits highly concurrent execution of operating system primitives and application programs (see Gottlieb and Kruskal [9]). Fetch-and-add is essentially an indivisible add to memory; its format is F&A(X,e), where X is an integer variable and e is an integer expression. The operation is defined to return the (old) value of X and to replace X by the sum $V+e$. Moreover, concurrent fetch-and-adds are required to satisfy the serialization principle enunciated above. Thus fetch-and-add operations simultaneously directed at X would cause X to be modified by the appropriate total increment while each operation yields the intermediate value of X corresponding to its position in this order. The following example illustrates the semantics of fetch-and-add: Consider several PEs concurrently executing F&A(I,1), where I is a shared variable used to index into a shared array. Each PE obtains an index to a distinct array element (although one cannot predict which element will be assigned to which PE), and I receives the appropriate total increment.

Fetch-and-add is a special case of Gottlieb and Kruskal's more general fetch-and-ϕ operation (where ϕ may be an arbitrary binary associative operator) [9]. Both of the classic test-and-set and compare-and-swap synchronization operations are also special cases of fetch-and-ϕ and the familiar load and store operations are degenerate cases. For example, Test&Set(S) is just Fetch&OR(S,*TRUE*).

1.3. The Power of Fetch-and-add

Using the fetch-and-add operation we can perform many important algorithms in a completely parallel manner, i.e. without using any critical sections. For example, as indicated above, concurrent executions of F&A(I,1) yield consecutive values that may be used to index an array. If this array is interpreted as a (sequentially stored) queue, the values returned may be used to perform concurrent inserts; analogously F&A(D,1) may be used for concurrent deletes. The complete queue algorithms contain checks for overflow and underflow, collisions between insert and delete pointers, etc. (see Gottlieb *et al.* [10]). We are unaware of any other completely parallel solutions to this problem. To illustrate the nonserial behavior obtained, we note that given a single queue that is neither empty nor full, the concurrent execution of thousands of inserts and thousands of deletes can all be accomplished in the time required for just one such operation.

2. Hardware Realization

As indicated above, our computational model is not physically realizable, due to fan-in (and other) limitations. Furthermore, memory modules operate sequentially; only one load or store may be satisfied in one cycle. If concurrent fetch-and-add or load operations were to be serialized at the memory of a real parallel computer, we would lose the advantage of parallel coordination algorithms, having merely moved the critical sections from the software to the hardware.

In fact, a parallel processor closely approximating our idealized paracomputer can be built. The NYU Ultracomputer uses a message switching network with the topology of Lawrie's [15] Ω-network (equivalently, the SW Banyan of Goke and Lipovsky [7]) to connect $N = 2^D$ autonomous PEs to a central shared memory composed of N memory modules (MMs). Note that the direct single cycle access to shared memory characteristic of paracomputers is approximated by an indirect access via a multicycle connection network.

Figure 1 gives a block diagram of the machine. The remainder of this section sketches the design of the processors, memory modules, and caches (see [4,8] for a more detailed description). The connection network is described in subsequent sections.

The Ultracomputer design places few constraints on the processors and memory modules; for example we take no stand on the RISC-CISC debate. Naturally, the fetch-and-add instruction is needed. In addition, the presence of a high bandwidth memory having non-negligible latency strongly favors processors that permit prefetching of instructions and operands. Although issued by the processor, fetch-and-add operations are effected in the MMs. When F&A(X,e) reaches the MM containing X, the value of X and the transmitted e are brought to the MM adder, the sum is stored in X, and the old value of X is returned through the network to the requesting PE.

The impact of network latency may be reduced by implementing a cache with each PE, thereby permitting single-cycle access to frequently-used instructions and data and reducing network traffic. Unfortunately, caching of read-write shared variables presents a coherence problem among the various caches. An obvious method of eliminating this problem is to simply not cache read-write shared variables and have the software distinguish between shared and private variables, typically by grouping them into separate memory-management segments. A more elaborate scheme is based on the observation that if, during a particular code segment, a shared variable is accessed read-only, or accessed only privately, then

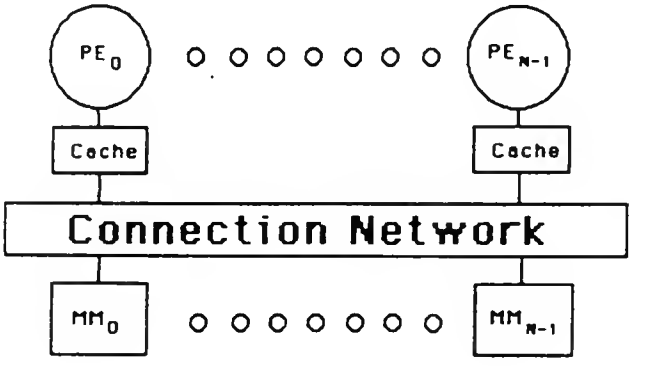

Fig. 1 – Ultracomputer Architecture

this variable may be cached during execution of that segment [16].

3. Network Design

The manner in which an Ω-network can be used to implement memory loads and stores is well known and is based on the existence of a (unique) path connecting each PE−MM pair. This section describes the overall design of the network while the subsequent section focuses on the individual switching nodes.

3.1. Combining Memory Requests

When concurrent loads and stores are directed at the same memory cell and meet at a switch, they can be combined without introducing any delay (see Klappholz [13] and also [10]). Combining requests reduces communication traffic and thus decreases the lengths of the queues within the switches, leading to lower network latency (i.e. reduced memory access time). Since combined requests can themselves be combined, the network satisfies the key property that any number of concurrent memory references to the same location can be satisfied in the time required for one central memory access. It is this property, when extended to include fetch-and-add operations, that permits the bottleneck-free implementation of many coordination protocols.

Since fetch-and-add is our sole synchronization primitive and is also a key ingredient in many algorithms, concurrent fetch-and-add operations will often be directed at the same location. Thus, as indicated above, it is crucial that a design supporting large numbers of processors not serialize this activity (see Pfister and Norton [17]). Enhanced switches permit the network to combine fetch-and-adds with the same efficiency as it combines loads and stores. When two fetch-and-adds referencing the same shared variable, say F&A(X,e) and F&A(X,f), meet at a switch, the switch forms the sum $e+f$, transmits the combined request F&A(X,$e+f$), and stores the value e in its local memory. When the value Y is returned to the switch in response to F&A(X,$e+f$), the switch transmits Y to satisfy the original request F&A(X,e) and transmits $Y+e$ to satisfy the original request F&A(X,f). Assuming that the combined request was not further combined with yet another request, we would have $Y=X$; thus the values returned by the switch are X and $X+e$, thereby effecting the serialization order "F&A(X,e) followed immediately by F&A(X,f)". The memory location X is also properly incremented, becoming $X+e+f$. If other fetch-and-add operations updating X are encountered, the combined requests are themselves combined, and the associativity of addition guarantees that the procedure gives a result consistent with the serialization principle.

3.2. Message Transmission

To ensure adequate network performance in large systems, a major goal in the design of the network is to attain a bandwidth proportional to the number of PEs. This has been achieved by use of the following techniques:

- Queues are associated with each switch to allow concurrent processing of requests for the same port whenever possible. The alternative adopted by Burroughs [12] of killing one of the two conflicting requests limits bandwidth to O(N/log N); see [14].
- Paths through the network are not maintained while awaiting memory responses. Thus, the interval between messages is the switch cycle time, rather than the network transit time.
- Flow control information is computed and transmitted in parallel with messages.

A major constraint on network performance is the delay inherent in off-chip communication between VLSI switching nodes, rather than the rate at which information can be processed within each node. Therefore, significant amounts of logic can be added to each node with advantage when that logic would help avoid global signaling and reduce bottlenecks within the network.

The number of chips required to implement each switching node is determined mostly by the high pin count required at each node, rather than the silicon area of the switching logic. Therefore, messages must be split into multiple packets and one of two methods can be used to transmit these packets through the network. The first is a bit-sliced implementation in which different components are handling different packets of one message (transmission of messages is "space-multiplexed"). Or the transmission of successive packets of a message can be time-multiplexed to the same component.

Space-multiplexing provides a higher bandwidth than time-multiplexing at the expense of more components. However, a large amount of "horizontal" communication and coordination must then take place between the different components of a switch, as routing and combining decisions have a global effect. This further increases both the complexity of such implementation and the switch cycle time. For MOS technologies, the off-chip delays impose an especially high overhead.

Several cycles are required to transmit each message if time-multiplexing is used. However, the internal logic of the switch can be pipelined so that messages can be handled on a per-packet basis and do not have to be assembled at each switch. Thus there can be as little as one cycle delay per switch for each request when queues are empty and hence time-multiplexing contributes an additive term to the delay rather than a multiplicative factor. However, queuing delays increase multiplicatively with the multiplexing factor, so that the performance of the network under heavy load may be seriously impaired [14]. In the current design we have chosen to use time-multiplexing, so that each message is divided into one packet containing the path descriptor, address and opcode, plus one or more data packets[2].

[2]At the expense of a severe increase in complexity, the address can also be transmitted in more

In addition, we assume that both the MM and PE numbers are transmitted. With additional internal switch complexity, the two D bit numbers can be transmitted as a single D bit amalgam [8].

The protocol used to transmit messages between switches is a message-level rather than packet-level protocol. That is, packet transmission cannot be halted in the middle of a message. A switch will accept a new message only if the available space in its queues guarantees that it will be able to receive the entire message.

4. Switch Structure

Each network switch is a 2x2 bidirectional routing device. The goals in the design of the switch are the following:

- Distinct data paths do not interfere with each other. Therefore, a new message can be accepted at each input port provided queues are not full. In addition, a message destined to leave at some output port will not be prevented from doing so by a message routed to a different output port.
- A packet entering a switch with empty queues when no other message is destined for the same output port leaves the switch at the next cycle.
- The capability to combine and de-combine memory requests should not unduly slow the processing of requests that are not to be combined.

This section begins with an overview of the entire switch and then discusses the protocols used for flow control. Finally, we describe the logic used to transmit responsed from the MMs to the PEs. This last description gives considerable detail since the material has not been published elsewhere.

4.1. Overview

Figure 2 shows a block diagram of a switching node. The "PE port" connects to either a PE or to an MM port of a preceding network stage and the "MM port" connects to either an MM or a PE port of a subsequent network stage.

Associated with each MM port is a combining queue capable of accepting a packet simultaneously from each PE port. Requests that have been combined with other requests are sent to a wait buffer at the same time as the combined request is sent to the MM port.

From each MM port a reply enters both the wait buffer associated with the MM port and the non-combining queue associated with the PE port to which the reply is destined. An associative look-up is performed in the wait buffer to determine if the reply was to a request that had been previously combined and, if so, the de-combined reply is sent to the non-combining queue at the appropriate PE port. Each non-combining queue has four inputs since messages may come from both MM ports and from both wait buffers.

---

than one packet [20].

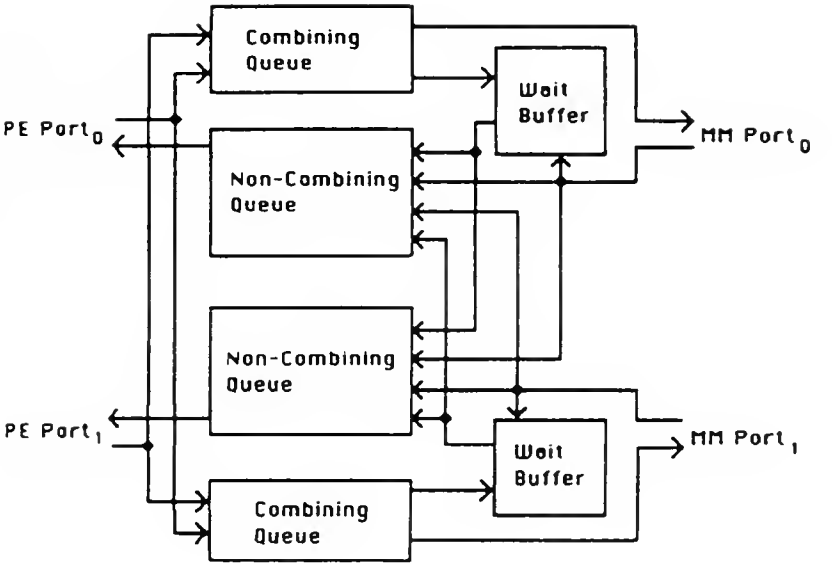

Figure 2 – Block Diagram of Switch

For packaging reasons, each switch is divided into a forward path component (FPC), consisting of the two combining queues, and a return path component (RPC) consisting of the wait buffers and non-combining queues. Data forwarded to a wait buffer from a combining queue are transmitted from the FPC to the RPC via ports called wait buffer output ports (WBOPs) and wait buffer input ports (WBIPs) on the FPC and RPC, respectively.

The combining queues used in the FPC are an enhancement of the VLSI systolic queue of Guibas and Liang [11] and are described in [2]. Further details on the design of combining queues can be found in [20]. The design of the RPC will be presented later in this section. For a detailed description of the implementation of a network for a planned 32-PE prototype, see [3].

4.2. Flow Control

The construction of the queues requires that there be an even number of packets per message and that switches distinguish even and odd cycles. At initialization, the parity of the cycle is the same as the parity of the stage to which the switch belongs, so that cycles that are even for a switch are odd for its predecessors and successors while the cycle parity of the FPC and RPC in the same switch are identical. Reception of messages starts only at even cycles while transmission of messages starts only at odd cycles.

Each port consists of data bits and two protocol bits: a data valid bit (DV) traveling in the same direction as the data and a data accept bit (DA) traveling in the reverse direction. In addition, input ports receive a routing (RO) bit whose value at the first cycle of a message transmission indicates to which output port the message is destined. The RO bit accompanying a packet entering a PE port at the

i-th stage of the network is the i-th most significant bit of the MM number; at an MM port and WBIP it is the i-th most significant bit of the PE number.

The two protocol bits, in conjunction with the RO bit, regulate the transmission of messages through the network. A sender asserts DV when it wishes to initiate a message transmission. Independently, a receiver asserts DA when it is able to accept a new message. A message transfer starts only if both DV and DA are asserted and the cycle parity is correct. Since these control signals are ignored during cycles when a message transfer cannot be started, they can be set ahead of time to overlap data transfer and flow control operations. Note that this is not strictly speaking a handshaking protocol: DA is not an answer to DV, nor an acknowledgment, but is issued independently and simultaneously. The sender is transmitting the data on the data lines whenever DV is asserted. If it receives DA, it assumes the data has been accepted and proceeds with the next packet. No provision for retry is necessary.

4.3. Return Path Component

The RPC routes responses from MMs to the requesting PEs. When a response to a request previously combined by the FPC is detected, the RPC will generate an additional response for the other requesting PE. Each RPC has two MM (input) ports (IPs) and two PE (output) ports (OPs). In addition, two wait buffer input ports receive information from the FPC.

An RPC contains two wait buffers, WB_0 and WB_1, one associated with each MM port and two four-input non-combining queues, which are implemented as eight single-input non-combining queues, Q_{ijk}, $0 \le i,j,k \le 1$. (To enable each input to accept a packet every cycle and to prevent a blockage of one output from interfering with the other output, one queue is required for each input/output pair where "inputs" include both wait buffer and input ports.) Queue Q_{0ij} is fed from IP_i and writes on OP_j; queue Q_{1ij} is fed from WB_i and writes on OP_j.

A message received on IP_i starting at cycle $2t$ with routing bit set to j is sent to Q_{0ij} and also to WB_j where its address packet is compared with the messages currently in the wait buffer. If a match is found, the wait buffer asserts its *match* line during cycle $2t+1$ but defers sending its generated response to Q_{1ij} until cycle $2t+2$ so that queues Q_{0ij} and Q_{1ij} receive the first packets of their messages at cycles of the same parity.

The DA signal can only be asserted at IP_i if there is at least one empty slot in each of the two queues Q_{0i0} and Q_{0i1}. In addition, there must be sufficient room in queues Q_{1i0} and Q_{1i1} for messages from WB_i corresponding to both the last message received at IP_i and the current message. Therefore, DA also cannot be asserted unless WB_i does not assert *match* and there is one empty slot in each queue Q_{1ix} or WB_i asserts *match* and there are two empty message slots in each[3] of these queues.

[3]Since the destination of the message in WB_i is known on-chip at this cycle, this test can be refined to require only two empty slots in the queue that is the destination of the message.

To arbitrate between the four queues Q_{xyk} that can send data to OP_k, the RPC keeps track of when each queue has last sent a message. Of the queues that are non-empty, the one having sent least recently is selected.

The DA signal is asserted at a wait buffer input port if the wait buffer will have an available slot to receive a message. As will be seen below, a slot in the wait buffer is capable of simultaneously receiving and transmitting a message. Therefore, DA will be asserted on $WBIP_i$ if WB_i either does not assert the *full* signal or asserts the *match* signal.

4.3.1. Wait Buffer

The wait buffer is an associative memory that stores information sent by the FPC when combining two F&A's into a single request. The wait buffer inspects all responses from MMs and searches for a response to a request previously combined by the FPC. When it finds a response to such a request, it generates a second response and deletes the request from its memory.

The structure of a wait buffer[4] (WB) is shown in Figure 3. A typical message slot is shown in the solid black box and consists of two registers (called Areg and Breg), compare logic, and a controller. Each register contains the data bits, a data valid (DV) bit, and, for the first packet of each message, a routing (RO) bit. The registers are connected in a loop of length two, and shift at each cycle. The Areg receives the address packet of a message at even cycles and the data packet at odd cycles. The opposite is true for Breg. Packets are stored in the format they are received from the WBIP with the RO bit appended to the address packet of each message.

Each slot connects to the following buses:

- The write bus (Wbus) is used to send data to the wait buffer from the FPC and connects to a wait buffer input port.
- The read bus (Rbus) is used by each slot for transmission of its message out of the wait buffer.
- The key bus (Kbus) contains the search key received from an MM port of the RPC.

The next-slot (NS) line is a one bit signal that is passed through all the slots in a daisy-chain fashion. It is used to select which slot will receive the next message from the FPC. Each slot computes

$$NS_{out} := NS_{in} \text{ and not } empty$$

and the end of this signal, which has passed through all the slots, is the *full* line of the wait buffer.

An adder is used to generate the second response to an F&A operation by summing the data packet received from a slot and the data packet received from an

[4]This structure requires each message to consist of a single address packet followed by a single data packet. Similar structures support messages containing a *fixed* even number of packets.

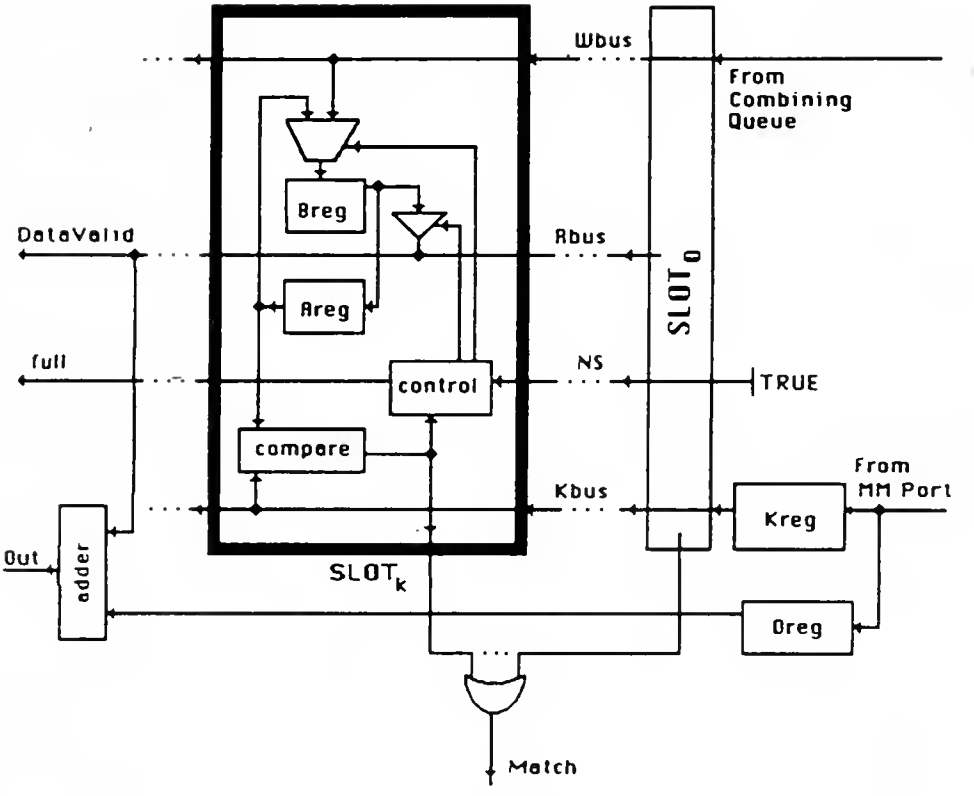

Figure 3 – Wait Buffer

MM. It passes address packets unchanged. Since the wait buffer is not required to respond until two cycles after it receives a matching memory response, the adder can take a full cycle.

The Dreg is connected to the MM port. It is loaded on odd cycles with the data packet of a message and presents that packet to the adder on the next cycle, which forwards the sum to the appropriate RPC queue on the cycle after that.

Each slot is *full* or *empty* depending on the DV bit in its Areg. For each pair of cycles $2t$ and $2t+1$, each wait buffer slot simultaneously performs the following operations:

- If DV is present on the Wbus, NS_{in} is on[5], and the slot is *empty*, the Areg is loaded with the address packet of the message at cycle $2t$. During cycle $2t+1$ the Breg receives the address packet and the Areg receives the data packet.
- If the slot is *full*, the data in the Areg and the data on the Kbus are compared during cycle $2t$. If the DV bit is present on the Kbus, the operation on the Kbus is an F&A, and the address and PE number are the same in the Areg as

[5]As a performance optimization, a slot can load its registers whenever it is *empty*. Its DV bit is then set from the logical AND of the DV bit on the Wbus and the NS_{in} signal.

on the Kbus, the slot asserts *match*. (Note that only one slot can detect a match because the combination of address and PE number uniquely identify each combinable request in the network [3].) If a match is detected, the slot will present its message on the Rbus at cycles $2t+1$ and $2t+2$. It will also be marked as *empty* during cycle $2t+1$ so that it can begin accepting a subsequent message at cycle $2t+2$.

If any slot asserts *match* during cycle $2t$, the wait buffer will assert *match* at cycle $2t+1$. The packet presented by a slot to the Rbus on cycles $2t+1$ and $2t+2$ will be presented to Q_{1lj} on cycles $2t+2$ and $2t+3$ after being processed by the adder.

5. VLSI Design Status

In preparation for the design of a complete combining switch chip, we have designed several chips which have been fabricated by DARPA's MOSIS facility.

We have received functional 11-bit wide 2x2 non-combining switch chips containing approximately 7500 transistors and fabricated in 3-micron NMOS. These parts operate at a clock speed of 23mHz with propagation delays from clock to output of approximately 25ns.. Power dissipation is approximately 1.5W. A 4x4 test network was constructed using four of these parts and functioned as expected.

We have also had a 6-bit wide portion of the FPC (without the adder) for a 2x2 combining switch fabricated in 4-micron NMOS. This switch is composed of four 1-input combining queues. These parts also operate as expected and have performance and power dissipation similar to the non-combining switches.

Since the final combining switches must be at least 32-bits wide and air-cooled, we have converted our design effort to the newly available scalable double-metal CMOS process, which promises minimum feature sizes as small as 1.6 microns. We have submitted, and are awaiting the fabrication of, a 35-bit non-combining switch using this CMOS technology.

We have also designed a fast 32-bit adder and plan to have a completed FPC by the end of the current academic year.

References

[1] A. Borodin and J.E. Hopcroft, "Routing, merging and sorting on parallel models of computation", in *Proc. 14th Annual ACM Symp. on Theory of Comp.*, May, 1982.

[2] S. Dickey, R. Kenner, M. Snir, and J. Solworth, "A VLSI Combining Network for the NYU Ultracomputer", *Proc. Intern. Conf. on Comp. Design*, 1985.

[3] S. Dickey, R. Kenner, and M. Snir, "An Implementation of a Combining Network for the NYU Ultracomputer", Ultracomputer Note #93, Courant Institute, New York University, 1985.

[4] J. Edler, A. Gottlieb, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P.J. Teller, and J. Wilson, "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach", *Proc. 12th Annual Intern. Symp. on Comp. Arch.*, pp. 126-134, 1985.

[5] J. Edler, A. Gottlieb, J. Lipkis, "Operating System Considerations for Large-Scale MIMD Machines", to appear in *Comp. in Mechanical Engineering*.

[6] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", *Proc. 10th ACM Symp. on Theory of Comp.*, pp. 114-118, 1978.

[7] L.R. Goke and G.J. Lipovsky, "Banyan Networks for Partitioning Multiprocessor Systems", *Proc. First Annual Symp. on Comp. Arch.*, 1973.

[8] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer", *IEEE Trans. Comp.* pp. 175-189, Feb. 1983.

[9] A. Gottlieb and C.P. Kruskal, "Coordinating Parallel Processors: A Partial Unification", *Comp. Arch. News*, pp. 16-24. Oct. 1981.

[10] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM TOPLAS* **5**, pp. 164-189, Apr. 1983.

[11] L.J. Guibas and F.M. Liang, "Systolic stacks, queues and counters", in *Proc. Conf. Advanced Research VLSI,* Jan. 1982.

[12] E.A. Hauck and B.A. Dent, "Burroughs' B6500/B7500 Stack Mechanism", *AFIPS 1968 SJCC*, pp. 245-251. Also in D.P. Siewiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 244-250.

[13] D. Klappholz, "Stochastically Conflict-free Data-base Memory Systems", *Proc. Intern. Conf. on Parallel Processing*, pp. 283-289, 1980.

[14] C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", *IEEE Trans. Comp.* **C-32**, pp. 1091-1098, 1983.

[15] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans. Comp.* **C-24**, pp. 1145-1155, Dec. 1975.

[16] K. McAuliffe, Ph.D. thesis, Courant Institute, New York University, 1985, in preparation.

[17] G.F. Pfister and V.A. Norton, " 'Hot Spot' Contention and Combining in Multistage Interconnection Networks", *Proc. 1985 Intern. Conf. on Parallel Processing*.

[18] J. T. Schwartz, "Ultracomputers", *ACM TOPLAS* **2**, pp. 484-521, 1980.

[19] M. Snir, "On Parallel Search", *Principles of Distributed Computing*, Aug. 1982.

[20] M. Snir and J. Solworth, "The Ultraswitch – A VLSI Network Node for Parallel Processing", Ultracomputer Note #39, Courant Institute, New York University, 1984.